RL-TR-97-210
In-House Report
March 1998

MULTI-PARADIGMATIC PROGRAMMING: A CASE STUDY

Michael L. McHale and Roshan P. Shah

AIR FORCE RESEARCH LABORATORY
ROME RESEARCH SITE
ROME, NEW YORK

RL-TR-97-239 has been reviewed and is approved for publication.

APPROVED:

JOHN J. CROWTER
Project Engineer

FOR THE DIRECTOR:

JOHN A. GRANIERO, Chief Scientist
Command, Control & Communications Directorate

DESTRUCTION NOTICE - For classified documents, follow the procedures in DOD 5200.22M. Industrial Security Manual or DOD 5200.1-R, Information Security Program Regulation. For unclassified limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

If your address has changed or if you wish to be removed from the Air Force Research Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

ALTHOUGH THIS REPORT IS BEING PUBLISHED BY AFRL, THE RESEARCH WAS ACCOMPLISHED BY THE FORMER ROME LABORATORY AND, AS SUCH, APPROVAL SIGNATURES/TITLES REFLECT APPROPRIATE AUTHORITY FOR PUBLICATION AT THAT TIME.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1998	In-House

4. TITLE AND SUBTITLE

MULTI-PARADIGMATIC PROGRAMMING: A CASE STUDY

5. FUNDING NUMBERS

PE - 61102F
PR - 2300
TA - C5
WU- 02

6. AUTHOR(S)

Michael L. McHale and Roshan P. Shah

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFRL/IFTB
525 Brooks Road
Rome, NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER

RL-TR-97-210

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTB
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-210

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Michael L. McHale/IFTB/(315) 330-1458

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This effort initiated an investigation of an extremely modular programming technique using multiple languages and programming paradigms. The investigation consisted of the implementation of a computer aided language learning program that uses definitions from semantically chosen vocabulary terms. The original problem was analyzed and divided into three modules: the interface, the word selector and the dictionary. Each of the modules (sub-programs) was handled using a different programming language and paradigm; procedural, logical and object-oriented. The end result was a small Windows based program with a strong artificial intelligence (AI) component. The system demonstrates the ease of development of multi-paradigmatic programs that combine AI techniques with a common, widely accepted type of interface. While this type of programming requires further studies, the preliminary assessment is that it is well suited to be the method of choice for future hybrid AI systems of medium size.

14. SUBJECT TERMS

artificial intelligence, computer programming, logic programming, computer-assisted language learning (CALL)

15. NUMBER OF PAGES

36

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

Table of Contents

List of Figures

Chapter 1

Introduction

Historically, computer programming has been divided into two camps. The first uses one language for every application regardless of the applicability of the features of that language to the problem. The second, the more productive method, chooses a language whose features most closely match the problem to be solved.

The first approach is typified by any number of general language communities, whose advocates consider their favorite language as the best choice (compromise) for power, speed and generality. While loyalty to a language can develop the ability of a programmer to handle a wider and wider array of problems that are difficult for the language, it can also lead to myopia. As Baruch observed, "If all you have is a hammer, everything looks like a nail."

The second approach selects a language, from those available, to do the task at hand. For manageable problems, with no surprises, this approach works well. But if the task has some out of the ordinary requirements then the group may lack the expertise in the language to accomplish the task. It is as if a carpenter is hired to replace a door. She has a "complete" tool set so is confident there will be no problems. But when she gets to the site she finds the current door is hung with square drive screws (who would have thought!) and she has no screw driver to extract them with.

Over the last couple of decades a number of techniques have been developed to overcome these shortcomings: structured and modular techniques help decompose a problem into more workable sub-problems; libraries have been pre-fabricated to obviate the need of recreating routines for difficult problems (i.e., problems with which the tool or language has trouble); and the importing of techniques from other programming paradigms (ex., the use of object oriented structures in BASIC).

This last technique has also given rise to languages that have been designed to combine two or more programming paradigms into a single language: loglisp (Robinson, Sibert and Greene 89); Leda (Budd 95); and C++ (Ellis and Stroustrup 1990) are examples of this. What these languages overlook is that much of the power of a language comes from its syntax. For example, the Definite Clause Grammar (DCG) syntax of Prolog allows one to write grammars in a very natural way. A DCG representation of a simple sentence might be:

sentence → noun_phrase, verb_phrase.

This is more than just "syntactic sugar". It allows the programmer to abstract away from the implementation and concentrate on the problem at hand; writing a grammar. When a language is designed to combine two or more paradigms under a single syntax this language specific type of abstraction is usually lost. Using the tool analogy, the combination of paradigms is like a Swiss Army knife. They may be a useful thing to carry but carpenters do not use them in place of a screwdriver.

So, while multi-paradigmatic languages are a mixed blessing, some of the other techniques mentioned above are more consistently useful. Modular programming, for example, is a good thing. But after having designed the separate modules why insist they all be programmed in the same language. What is really needed is a division of labor. When building a house we let a carpenter do the framing, a plumber do the plumbing and an electrician do the wiring. Why not use the same approach in programming. Divide the problem into modules and select the language best suited for each module. Then combine the modules to solve the original problem.

The traditional way of linking languages like this is through foreign function calls. This has been somewhat successful but is often inelegant for the programmer because the operating system (ex., DOS) does not provide an easy, graceful way to do this. That has changed with Microsoft Windows.

Microsoft Windows was designed as a friendly operating system that allows for the integration of disparate applications (ex., word-processors, spreadsheets, databases). Windows assists the integration through three mechanisms: Dynamic Data Exchange (DDE) which allows for the transfer of data between applications; Dynamic Link Libraries (DLL), a type of foreign function call that is linked dynamically to an application at run time; and Object Linking and Embedding (OLE), that allows the embedding of an object from one application inside another (ex., a spread sheet or graph inside a word-processor document). These mechanisms

allow for extremely modular programming and the seamless integration of applications. They also provide for the integration of disparate languages though that was not their original intent.

DLL in particular, can be used to integrate programs written in different languages. DLL can be viewed as executable "black boxes" (modules) that can be used by any program or application that is equipped to work with them. Indeed, much of the Windows programming interface (API) is built using DLL.

Windows, then, allows for the type of programming we desire to investigate. One based on the following process:

1) Analyze the problem and divide it into natural, meaningful sub-problems

2) Choose the language that best suits each sub-problem

3) Write the modules for each sub-problem

4) Integrate the modules into a program that solves the original problem.

Step 1 probably deserves much more attention than we give it here but for now we are assuming this step is doable. Step 3 is relatively easy. That is, given a tractable sub-problem that is well suited to a language makes for a fairly easy programming task. Likewise, step 4 is greatly facilitated by the Windows environment and various Windows-based software. We will discuss Step 2 in more detail even though it is not obviously difficult.

If the choice in step 2 came down to two related languages such as Pascal and C or Scheme and Lisp then the choice might well rest on factors other than suitability.

That is, if both languages are suitable to the task then the choice might rest on programmer preference, speed of execution or development time. For us, step 2 should probably be rephrased to read "choose the paradigm (and then the language) that best suits each sub-problem."[1] Each programming paradigm (ex., logical, functional, procedural) has its own radical view of the world. For an imperative language, like BASIC, a program is a step by step enumeration of the tasks to be performed. For a logical language, like Prolog, a program is an instantiation of variables that are true for a given pattern and constraints.

To program in the different paradigms requires a different mindset, different viewpoints, different representations. How important are the different representations for solving problems? Very. A good representation can make a difficult problem easy while a bad representation can make an easy problem nearly impossible. As an example, consider the following problem.

The MCB Problem

You are presented with two grids (Figure **1a** and **b**) that are identical except **b** has had two opposing corners removed. You are also given a large stack of tiles that are 2x1 in size. Can you completely cover both grids with the tiles so that there are no overlaps or overhangs? If yes, show each covering. If no, give a proof as to why they cannot be covered.

[1] Thus the term Multi-paradigmatic Programming (MPP).

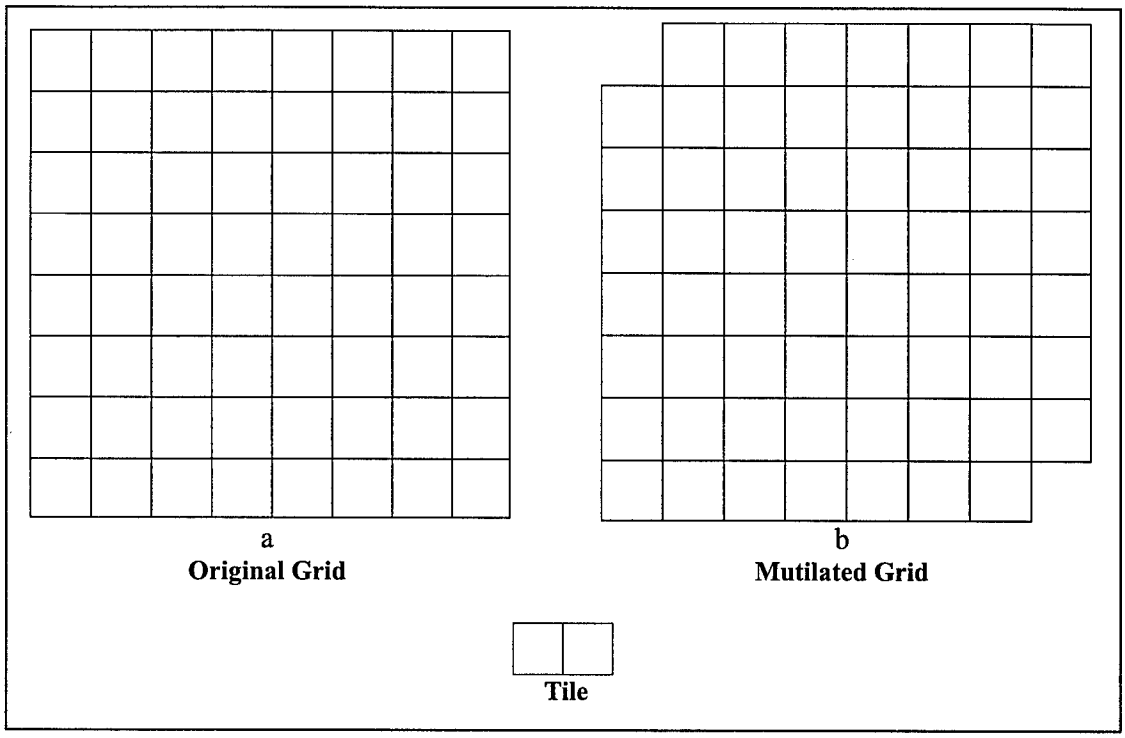

Figure 1. Original and Mutilated Grids.

Covering grid **a** is no problem. Grid **b** is not as obvious. One could try by hand for a while to see if there is an easy solution. Or a program could be written to try all possible layouts and either find one that works or thus prove there is none possible. But there is an easier solution; one based on a new representation. Instead of viewing the grids as simply grids, view them as checker boards (Figure **2a** and **b**). Covering **a** still remains trivial. Notice though that each tile you place on **a** covers two blocks of different color; one white and one black. Since there are 32 of each color it is trivial to place 32 tiles to completely cover the board. Now note that grid **b** has had two squares of the same color removed. This leaves 32 white squares and 30 black squares. Therefore, regardless of layout, after placing 30 tiles on grid **b** there are two white squares left uncovered. Since the tiles have to cover two squares of different colors, grid **b** cannot be covered. The representation makes it obvious.

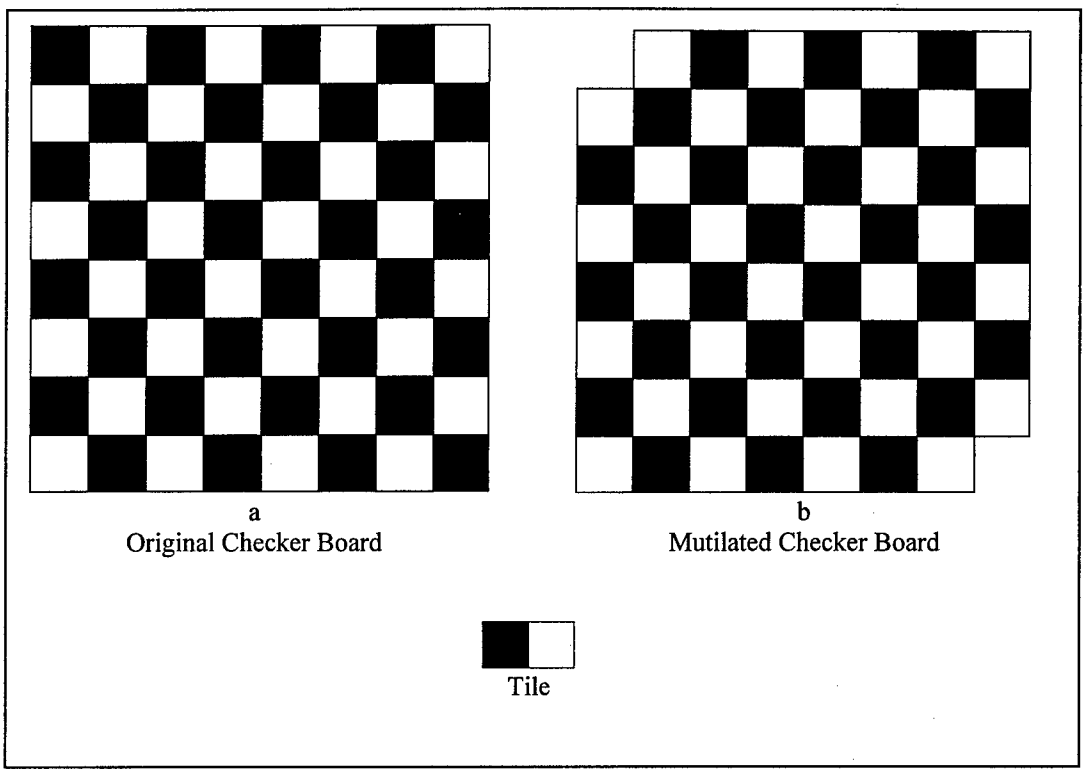

Figure 2. Original and Mutilated Checker Boards.

The representation, then, is paramount to the process of solving complex problems. The various representations also provide power to the MPP approach. Actually, there is something going on here that is more subtle than just the choice of representations. There is also style. Obviously, if a program requires matrices then one would want to write it in APL or some other language which handles matrices with ease. But, because of the different viewpoints that languages have of programming, many languages are shackled by what is considered "good" programming style in that language. A case in point is C. We recently encountered a book on programming style in C that advised never to write anything recursively.

This is a shame as C handles recursion quite nicely and some algorithms are recursive by nature.

For example, when we first encountered the Quicksort algorithm (due to C.A.R. Hoare) it was taught by a "dyed-in-the-wool" C programmer that believed in the non-recursive nature of C. The result was a presentation of the algorithm from a strictly iterative viewpoint.[2]

"Take a one-dimensional matrix of numbers and divide it in the middle. Then take two pointers and move out in each direction from the middle swapping those elements that ..."

The result was a program description that was nearly incomprehensible. How much easier it is from the recursive viewpoint.

"To quicksort a group of items divide the group into two; littles and bigs. (Where the smallest big is bigger than the biggest little). Then quicksort the bigs and quicksort the littles and put them back together into a single group."

```
quicksort(Group, SortedGroup) → divide(Group, Littles, Bigs),
                                quicksort(Littles, SortedLittles),
                                quicksort(Bigs, SortedBigs),
                                join(SortedLittles,SortedBigs,SortedGroup).
```

Figure 3. Prolog code for Quicksort.

[2] Though to be fair, Knuth's presentation of the algorithm (Knuth 73, Vol. 3, 114-123) was also iterative.

The recursive version is short, elegant and powerful. We wanted to investigate MPP by applying it to a real problem whose sub-problems could benefit from this variety of representations. The paradigms we were considering were logical, functional, procedural and object oriented. The ideal problem should be complex enough to benefit from a variety of approaches. For instance, it should require a user interface, database access, some search and perhaps some conflict resolution. After considering various problems we settled for class room scheduling.

Chapter 2

Problem Selection

Classroom scheduling is a complex, constrained resource allocation problem. The problem consists of scheduling the students, teachers and classrooms for all the subjects in a normal high school. Rather than just a single viewpoint, the problem has various viewpoints to consider. The students, for instance, have a minimum number of courses they have to take each year. Some courses are required (ex., English, Social Studies) while some are electives (ex., language, band). The teachers have to teach so many hours a day but must have time for breaks and lunch. They may also have preferences about when they teach certain courses. Satisfying the preferences is not critical but it is something that should be attempted. The classrooms are set in size and number and some are set for function (laboratories, band room, gym). The problem is driven by the students' required and elective courses and then constrained by class size and the teachers and classrooms that are available.

We analyzed the problem, decomposed it, assigned the various tasks to different languages, and began developing algorithms. Through this process we learned a great deal about problem decomposition and some of the criteria involved in language selection. Our time though, seemed to center on solving classroom scheduling and not on multi-paradigmatic programming. We decided to try a different task.

We selected a much easier problem but one that was still conducive to a multiple language approach. As with our first selection, we needed an unclassified domain that was widely and readily understood. We chose to do a computer assisted language learning (CALL) program for learning vocabulary.

This type of program was among the first computer aided instruction (CAI) programs. They have found a niche in helping motivated students learn items that require straight memorization; like foreign language vocabulary. In their standard form, the program selects four words along with their respective definitions. One of the words is then designated as the word to be learned (the target word). The target word and all four definitions are listed, and the student's task is to select the correct definition from the incorrect ones.

As presented, this program could easily be written in BASIC, C, Pascal, or any other general purpose language. Indeed, thousands of them have been. There is, however, a shortcoming with the program as described so far. This type of program is helpful only when used by motivated students – as the programs very quickly becoming boring. There are two reasons for this. The first is that the task is highly repetitious; read the word, read the definitions, pick one and see if it's correct. This cycle becomes old within minutes. The second reason for the boredom is that picking the right definition is often not a challenge because the alternative definitions are poorly selected. The usual (traditional) method of choosing the other three choices from the dictionary is to do it randomly, but as Figure 4 shows this may not be

sufficient. The figure gives the target word toreador and four definitions, all for nouns. The student may have no idea what a toreador is, but given that it ends in or knows that it is probably a person of some sort (ex., actor, doctor, governor) or a machine (ex., compressor, monitor, projector). From this, it is simple to guess the correct answer as being #2.

Toreador

1) The fact of having the skill, power, or other qualities that are needed in order to do something.[3]
2) A man who takes part in a Spanish bullfight, especially one riding on a horse.
3) A narrow arm of the sea between cliffs or steep slopes, especially in Norway.[4]
4) The fur of the coypu.[5]

Figure 4. Randomly Selected Choices.

A better way to choose the alternative answers is to use semantically related words. This approach is more of a challenge to the student and thus helps maintain interest. It also provides the added benefit of aiding memorization by clustering related terms. In the example above, this approach might choose the definitions for commando, gaucho and sniper (see Figure 5).

1) A man who takes part in a Spanish bullfight, especially one riding on a horse.
2) (a member of) a small fighting force specially trained for making quick attacks into enemy areas.
3) A cowboy of the South American pampas.
4) One who shoots at individuals, especially enemy soldiers, from a concealed or distant position.

Figure 5. Semantically Selected Choices.

[3] ability
[4] fjord
[5] nutria

This significantly reduces the ability to easily eliminate the alternative answers and reinforces the learning of the words by forcing the student to differentiate similar terms.

However, if the semantically related definitions are pre-selected at program design time then the amount of work for the program designer increases dramatically. That is, the designer not only has to acquire the vocabulary (the words and their definitions) but has to determine which words can accompany each word and explicitly mark them. This could be done be developing groups of related words (ex., toreador, commando, gaucho, sniper) or more generally by creating a semantic network of the words.

We had no desire to spend our time creating a semantic network or a large lexicon for the program. We did not need to as we already had both available.

Chapter 3

A Multi-Paradigmatic Approach

The CALL system (WordGenius) is comprised of three modules: the interface; the word and definition selector; and the dictionary. The complete system could have been written in Visual C++ or some other general purpose language but we opted for a multiple language, multiple paradigm (MPP) approach. The interface was written in Visual Basic (object oriented and procedural), the word selector was written in Prolog (logical) and the dictionary came from a pre-existing language resource. We decided not to modify the dictionary in any way because it was already represented in a semantic network. That left us with two programmers, two sub-programs and two languages. Each programmer could then use the language in which they felt most comfortable to write a sub-program that the language could easily handle.

The Implementation

We will discuss each of the modules below. We will start with the dictionary since it was not modified at all.

The Dictionary

The dictionary we use is the noun.dat file from WordNet (Miller 93). The data consists of more than 87,000 nouns arranged in synsets (sets of synonyms) with each synset having a common definition. The data is further arranged

hierarchically for hyponymy (is-a relations) and meronymy (has-a or has-part relations). The file itself is a binary file arranged by byte-offset. The format of the data file consists of more than 60,000 lines, the form of which is shown here (WNDB 93).

synset-offset lex# pos w_cnt word id [word id ...] p_cnt [ptr ...] gloss

Where

synset-offset	the current byte offset in the file (8 digits)
lex#	the lexicographer file from which the data was taken (2 digits)
pos	part-of-speech (**n** for nouns)
w_cnt	the number of words in the synset (2 digit hexadecimal)
word	ASCII form of the word (variable length)
id	the sense of the word (1 digit hexadecimal)
[...]	optional repetitions (as many as needed)
p_cnt	the number of pointers from the synset (3 digits)
ptr	a list of pointers from the synset (see below)
gloss	a definition, an example sentence or both (variable length)

A pointer (**ptr**) consists of a pointer symbol followed by a space, the synset-offset of the target synset, followed by a space, a **pos** to indicate to which data file the offset refers, followed by a space and a four digit source/target field that indicates which sense of the synset (source and target) the offset refers.

The pointer symbol for nouns are:

!	Antonym
@	Hypernym
~	Hyponym
#m	Member meronym
#s	Substance meronym
#p	Part meronym
%m	Member holonym
%s	Substance holonym
%p	Part holonym
=	Attribute

For instance, the entry for landing is:

00026059 04 n 01 landing 0 006 @ 00023747 n 0000 ~ 00029218 n 0000 %p 00158083 n 0000 ~ 00171746 n 0000 ~ 00171849 n 0000 ~ 00172390 n 0000 | the act of coming to land after a voyage

This can be understood as:

00026059	the line starts at the 26,059th byte of the file
04	it was taken from the fourth file
n	it is a noun
01	there is 1 member of the synset
landing	the word is landing
0	this is the first sense of the word
006	there are 6 pointers
@ 00023747 n 0000	the superordinate of landing (arrival) is at byte 23747, it is a noun and the pointer refers to all the words of this offset and that one
~ 00029218 n 0000	a subordinate of landing (debarkation) is at byte 29218, it is a noun ...
%p 00158083 n 0000	a (has-part) subordinate of landing (landing approach) is at 158083, it is a noun ...
~ 00171746 n 0000	a subordinate of landing (touchdown) is at byte 171746, it is a noun ...
~ 00171849 n 0000	a subordinate of landing (aircraft landing) is at byte 171849, it is a noun ...
~ 00172390 n 0000	a subordinate of landing (splashdown) is at byte 172390, it is a noun ...
| the act of ...	the definition of landing is the act of ...

We use the IS-A relations for the definition selection. These relations are a representation of a semantic network but one that is local in nature. To find where any given word fits into the network it is necessary to follow the parent links back to one of the dozen or so top elements in the network. This is not a concern for this particular program as we only need the local links to find siblings. If we were to expand the program so that the word selection was determined by the user (i.e., for

topic) then we would need to determine that the word selection was from the correct part of the hierarchy.

The program, as written, is of marginal use because it does not constrain the word selection to topic. WordNet has 87,000 nouns many of which would be unknown to most users. The semantic hierarchy in WordNet would allow limiting choices to one domain and this is probably the only way the program could productively be used. We chose to use WordNet because of the existing semantic hierarchy and because its size demonstrates nicely the ability to "scale up" to a real domain. Also, as can be seen above, the lines of the noun file are structured but variable thus making them perfect candidates for parsing with Prolog.

Word Selection

Word selection is done by Prolog. The target words that are used are chosen randomly but the alternative definitions for the word are siblings of the target word in the WordNet IS-A hierarchy. Only slightly more than four thousand of the 60,000 lines in the dictionary have four or more children. That means that this process of selection only uses one out of fifteen possible words as a target word. Were this anything but a demonstration program, that would be a serious limitation.

The random selection process does not demonstrate any intelligence in the selection of words but rather was chosen because it was easy to do and demonstrates the use of semantically related words for this type of program. If a word does not have three or more siblings we should allow other relations

(ex., descendents of siblings, cousins) as long as the resulting words are not synonyms of the target word. The selection process needs to know both the structure and size of the dictionary.

The program, as implemented, selects a random number less than the dictionary size (in this case 9 billion bytes), opens the dictionary for binary reading, sets the pointer to the random location (random number of bytes) and reads to the end of the line in which it finds itself. Having reset the pointer, the next line is then read and parsed for input. This process (selection of a random location, the reading and parsing of a line) continues until a line with at least four children is found. Then, one of the children is selected as the target word and its siblings are selected as alternative definitions. The words and definitions are returned to the Visual Basic program.

With the exception of parsing the lines, none of the above is very logical in nature; not very Prolog like. We are sensitive to that fact but there are two reasons why Prolog was chosen for all these tasks.

1) The Prolog we are using, Amzi!, has built-in predicates that perform all the functions needed (i.e., binary file I/O).[6]
2) On average, 15 random numbers have to be generated to find a line with four children. It was easier and faster to have Prolog backtrack into the random

[6] Amzi!'s file I/O is actually implemented in C++.

number generator than it was to have Visual Basic loop over the call to Prolog.

```
line → file_info, word_info, word_group, ptr_group, gloss.
        file_info → offset, lex_file.
                offset → {number}.
                lex_file → {hex number}.
        word_info → n, cnt.
                n → {n}.
                cnt → {hex number}.
        word_group → [].
        word_group → word, id, word_group.
                word → {alphabetic}.
                id → {hex number}.
        ptr_group → [].
        ptr_group → p_cnt, ptr_groups.
                p_cnt → {number}.
                ptr_groups → [].
                ptr_groups → ptr, ptr_groups.
                        ptr → ptr_symbol, offset.
                                Ptr_symbol → {!|@|~|#m|#s|#p|%m|%s|%p|=}
        gloss → gloss_symbol, {string}.
                gloss_symbol → {|}.
```

Figure 6. Pseudo-Prolog DCG.

The Interface

The interface and program control were written entirely in Visual Basic. The interface consists of two screens; the timer screen (Figure 7) and the question screen (Figure 8). When program execution begins, the timer screen is displayed and the student has the option of having the program run continuously or have it run at set intervals. The interval option is convenient when the student wishes to interact with the program over an extended period but has other work that also needs doing. It also allows for interaction over long periods without the program becoming boring.

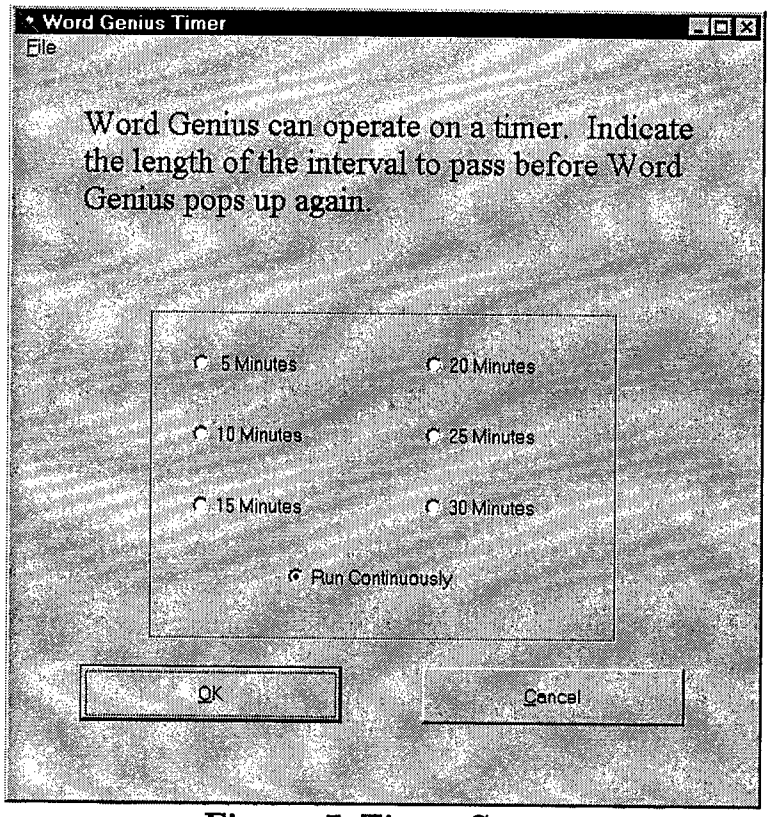

Figure 7. Timer Screen.

Once the student makes a timer selection, the Visual Basic program initializes the Prolog program and requests a target word and associated definitions. This is done using the Amzi! Prolog Logic Server (AMZI4.DLL) and an extension for Visual Basic that is supplied with Amzi!. This is a very simple to use interface between the languages. Figure 9 shows the basic code for initializing the Logic Server.

The Visual Basic code for word selection (Figure 10) is also straightforward. Visual Basic calls the Prolog procedure main which does the word and definition selection, and asserts them into the Prolog knowledge base. Then each word and definition is called separately using the procedure word(X,Y). It would be possible to

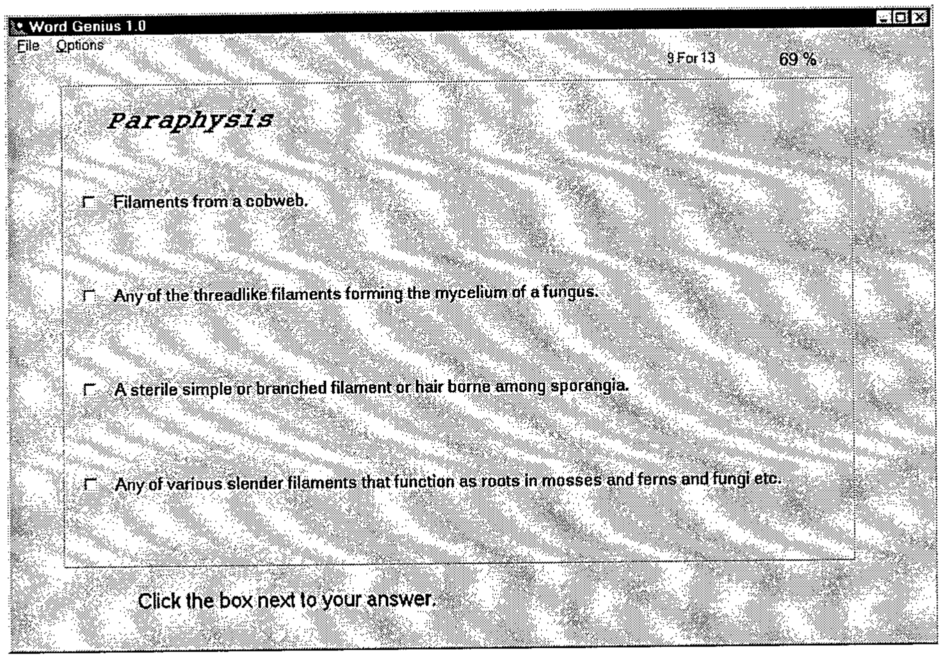

Figure 8. Question Screen.

```
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'' Main Form Handler Starts up Amzi! Prolog
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

sub form_load()
   Dim rc As Integer, tf As Integer
   Dim Term As Long
   Dim xplname As String

   ' Setup our xpl pathname
  xplname = App.Path + "\FLASH.XPL"

   ' Initialize the runtime and load FLASH=.XPL, which contains
   ' all the rules and expertise for this application
   InitLS (xplname)
   LoadLS (xplname)

End Sub
```

Figure 9. Visual Basic code for Prolog Initialization.

pass all four words and definitions as parameters rather than assert them and then collect them. The former would be cleaner from the Prolog side but perhaps a bit more unwieldy from the Visual Basic side.

```
Sub Words()
''''''''''''''''''''''''''''''''''''''''''''''''
' Display All the Words
''''''''''''''''''''''''''''''''''''''''''''''''
   Dim rc As Integer, tf As Integer
   Dim Term As Long

 'Run main first
   tf = CallStrLS(Term, "main")

' Issue the Prolog query:  word(X,Y)
   tf = CallStrLS(Term, "word(X,Y)")

   ' Loop through all the words
   While (tf = True)
     Call GetArgLS(Term, 1, bSTR, PWord)
     Call GetArgLS(Term, 2, bSTR, PDefinition)
     Call AssignWords(PWord, PDefinition)
     tf = RedoLS()
   Wend

End Sub
```

Figure 10. Call Word Selection.

After the word selection has been accomplished, the target word and four definitions are displayed, as was shown in Figure 8. The student selects one of the four definitions by clicking on it with the mouse. If the correct answer was selected (Figure 11) it is highlighted (in bold) and the other choices are deactivated. The words corresponding to the alternative answers are displayed above each definition. If the correct answer was not selected (Figure 12) then the correct definition is again highlighted in bold while the definition the student selected is highlighted in red. The words corresponding to the alternative answers are again displayed above each definition.

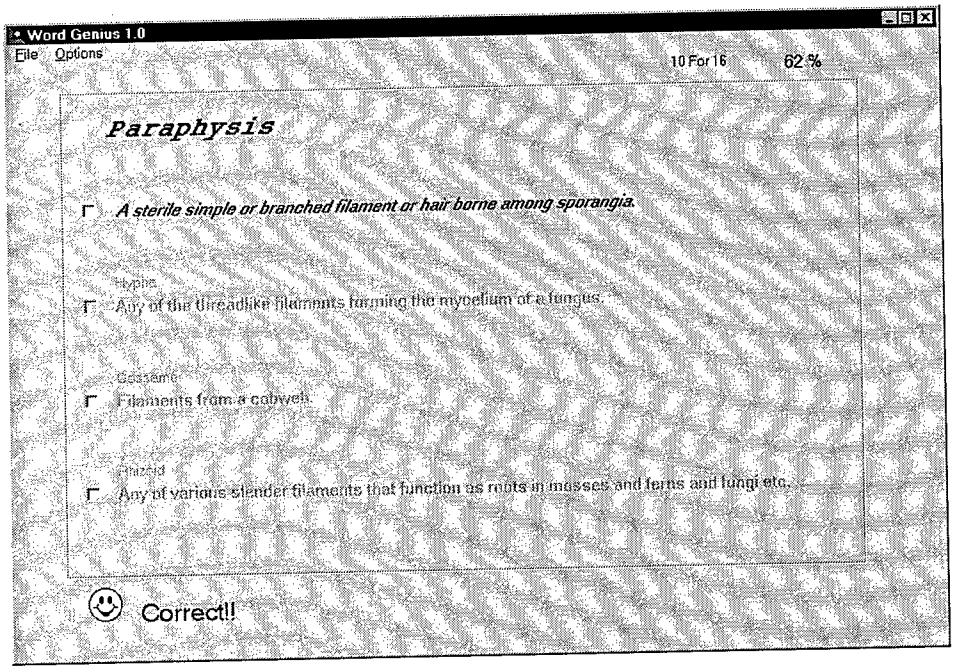

Figure 11. The Correct Answer was Selected.

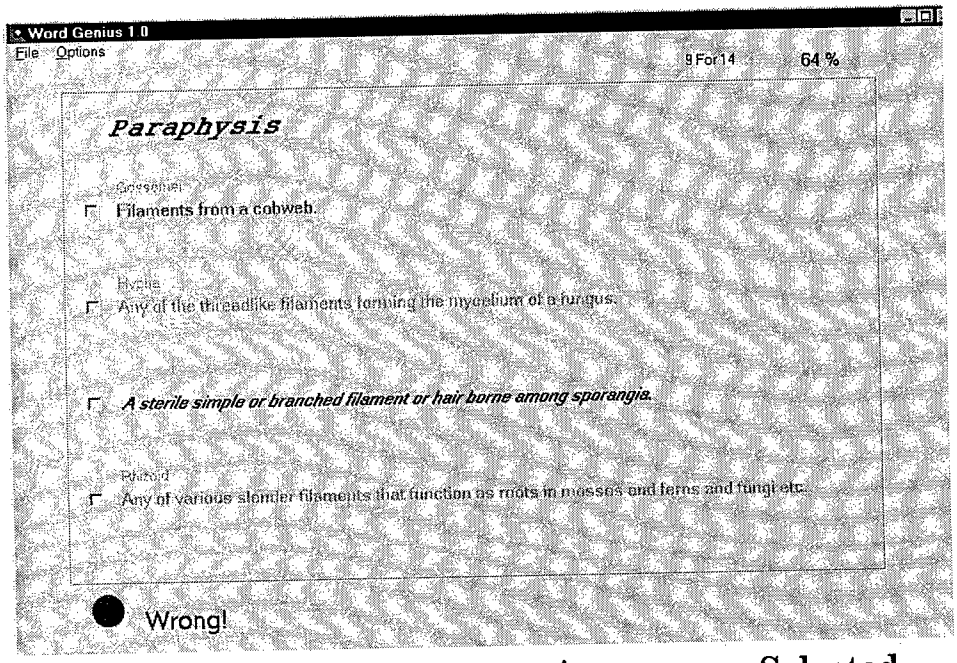

Figure 12. An Incorrect Answer was Selected.

When the student has read the correct answer they can click anywhere on the window to either get the next question or start the timer. The timer runs in the background and uses very little computer resources. Notice too that the program keeps track of how many questions have been answered and the number and percentage that have been answered correctly.

Extensions and Testing

We have tested the system with four different dictionaries: WordNet, as described above; a subset of WordNet consisting of just the shorter words; a hand-coded dictionary of English vocabulary (approximately 200 words); and a hand-coded bilingual dictionary of Tagalog (Filipino) terms and English definitions. No changes to the interface were required for the various dictionaries. The only changes to the Prolog engine that were required was a simplification of the reading predicates (for text files instead of binary) and a modification to the sibling choices. These changes took less than one hour to complete.

With minor modifications to the interface one could write a program that allowed the student to choose the file desired as long as the files used some line structure similar to

```
w(index, relation, word, definition)
```

where relation was either a list of allowable siblings or the location in a semantic hierarchy. The dictionaries, then, would become basically data files for the Prolog program.

Chapter 4

Summary

We have investigated the use of multi-paradigmatic programming (MPP) techniques for a knowledge intensive, interface intensive task. The use of multiple languages from different paradigms to create the program proved to be efficient and versatile.

Obviously, doing a program's interface in the object oriented Visual Basic makes much more sense than doing it in Prolog or some other language less suited for user interface. Thus the question arises, "why not do the entire program in Visual Basic?" The answer, though, follows the same logic: another language, like Prolog, can handle the background processing better than Visual Basic. By exploiting individual languages for their strengths, programming applications becomes easier and faster, and the applications themselves improve in speed and capability.

MPP involves some overhead in the initial program design as it requires substantial decomposition of the problem (more so than single language approaches). This overhead is more than offset by the power and versatility of the MPP approach. The power and versatility may come with too much cost for single programmers but should be highly useful for team (i.e., larger software) projects.

Since MPP is an extension of modular programming, it reaps the benefits of that approach. A project leader can divide the tasks so that task experts (ex., interface designers) can deal directly with programmers that deal mainly with their task

(ex., Visual Basic programmers). The programmers are dealing only with tasks that their language was designed to do. This makes their job easier as they are not trying to fit square pegs into round holes.

This last point cannot be overemphasized. Each programmer in an MPP project gets to write "pure", or at least nearly "pure", code. For instance, in Prolog it would be possible to pass all parameters as arguments so that there would be no extra-logical predicates; not even a print statement.

Our choice of Prolog for this project was not entirely arbitrary. It was chosen largely because it naturally complements Visual Basic. Prolog language developers, like developers of other languages used for artificial intelligence (i.e., lisp, logo, scheme, smalltalk) have traditionally emphasized the handling of knowledge representations – not interface development. Thus, most of them would be good candidates for MPP. If our experience with MPP is a fair indication, then MPP may well be the future for medium sized artificial intelligence programs.

Bibliography

Budd, T. (1995) *Multiparadigm Programming in Leda.* Addison-Wesley, Reading, MA.

Ellis, M.A. and **B. Stroustrup** (1990) *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA.

Krumm, R. (1994) *Making Windows Applications Work Together.* M&T Books, New York, NY.

Miller, G. (1993) *Five Papers on WordNet.* Available from ftp://ftp.cogsci.princeton.edu/pub/wordnet/5papers.ps

Knuth, D.E. (1973) *The Art of Computer Programming. Volume 3. Searching and Sorting.* pp. 114-123. Addison-Wesley, Reading, MA.

Robinson, J.A., E.E. Sibert and **K.J. Greene** (1989) *The Loglisp Programming System.* RADC Technical Report 85-89. Rome Air Development Center, GAFB, NY.

Telles, M.A. (1994) *Mixed Language Programming.* M&T Books, New York, NY.

WNDB (1993) *The WordNet database.* Available from http://www.cogsci.princeton.edu/~wn/doc/man/wndb.5WN.html

*U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61174